NASA/CR—2000-209416

A Simulation Study of Paced TCP

Joanna Kulik, Robert Coulter, Dennis Rockwell, and Craig Partridge
BBN Technologies, Cambridge, Massachusetts

January 2000

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the Lead Center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA's counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized data bases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA Access Help Desk at (301) 621-0134

- Telephone the NASA Access Help Desk at (301) 621-0390

- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076

NASA/CR—2000-209416

A Simulation Study of Paced TCP

Joanna Kulik, Robert Coulter, Dennis Rockwell, and Craig Partridge
BBN Technologies, Cambridge, Massachusetts

Prepared under Contract N00600–92–C–3377

National Aeronautics and
Space Administration

Glenn Research Center

January 2000

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076
Price Code: A03

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22100
Price Code: A03

A Simulation Study of Paced TCP

Joanna Kulik, Robert Coulter, Dennis Rockwell, and Craig Partridge
BBN Technologies *
Cambridge, Massachusetts

Abstract

In this paper, we study the performance of paced TCP, a modified version of TCP designed especially for high delay-bandwidth networks. In typical networks, TCP optimizes its send-rate by transmitting increasingly large bursts, or windows, of packets, one burst per round-trip time, until it reaches a maximum window-size, which corresponds to the full capacity of the network. In a network with a high delay-bandwidth product, however, TCP's maximum window-size may be larger than the queue size of the intermediate routers, and routers will begin to drop packets as soon as the windows become too large for the router queues. The TCP sender then concludes that the bottleneck capacity of the network has been reached, and it limits its send-rate accordingly. Partridge proposed paced TCP as a means of solving the problem of queueing bottlenecks. A sender using paced TCP would release packets in multiple, small bursts during a round-trip time in which ordinary TCP would release a single, large burst of packets. This approach allows the sender to increase its send-rate to the maximum window size without encountering queueing bottlenecks. This paper describes the performance of paced TCP in a simulated network and discusses implementation details that can affect the performance of paced TCP.

1 Introduction

TCP was originally designed to run over a variety of communication links, including wireless and high-bandwidth links. Although researchers understand a great deal about TCP's strengths and weaknesses in the networks of today, recent technological advances in satellite and fiber-optic networks have caused them to re-evaluate TCP's purported flexibility. They have already identified some major obstacles to the use of TCP in these networks. One problem that is common to both satellite and fiber-optic networks is that the capacity of these networks, determined by the product of the bandwidth and the delay of the network, can be more than ten times greater than in conventional networks. The mismatch between the high capacity of these networks and available storage at the intermediate routers in the network poses unique problems for TCP. Given an ever-growing interest in these new network technologies and the centrality of TCP in network communication, further investigation of obstacles to the use of TCP in high delay-bandwidth networks is a matter of pressing importance.

This paper examines paced TCP, a modified version of TCP designed to overcome the problems of queueing bottlenecks in a network with a high delay-bandwidth product. In a typical network, TCP optimizes its send-rate by releasing increasingly large bursts, or windows, of packets, one burst per round-trip time, to the receiver until it reaches its maximum window-size, at which point it has reached the full capacity of the network. In a network with a high delay-bandwidth product, however, TCP's maximum window-size may be larger than the queue capacity of some of the network's intermediate routers. Larger windows overload such router queues, and the routers begin to drop packets. TCP interprets dropped packets as congestion at the bottleneck, even though no congestion is present. A single sender running TCP has simply released too many packets into the network in too short a time:

*This work would not have been possible without the additional contributions of Tim Shepard and Koling Chang.

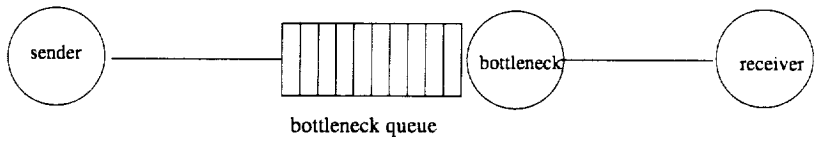

Figure 1: Topology for a TCP connection running over a single bottleneck-link.

Parameter	Low-delay network	high-delay network	
		(large queue)	(small queue)
Round-trip Delay	50ms	500ms	500ms
Bottleneck bandwidth	1.5Mb	1.5Mb	1.5Mb
Packet size	1024 bytes	1024 bytes	1024 bytes
Bottleneck queue length	10 packets	100 packets	10 packets
Network capacity/ TCP maximum window	9 packets	91 packets	91 packets

Table 1: Setup for different TCP connections running overa single bottleneck link.

the queue cannot handle the overload; and TCP detects its self-produced queueing bottleneck. The disparity between the total capacity of a high delay-bandwidth network and the capacity of individual queues in the network is enough to make TCP's algorithms break down.

Consider the three concrete networks whose characteristics are listed in Table 1 and depicted in Figure 1. The first is a low-delay network; its round trip times are 50ms. The second and third networks have high delays; round trip times for these networks are typical for a satellite network, 500ms.

Figure 2 gives performance characteristics over time in the low-delay network. This performance is a function of the bottleneck bandwidth, the network delay, and the packet size. In this network, as in most low-delay networks, the capacity of the bottleneck queue is roughly equal to the capacity of the network itself. The sender running TCP begins in slow-start mode, with an initial window of 2 packets. During each RTT, the sender receives a new acknowledgment and increases its window by 1 packet. As a result, TCP releases an exponentially increasing number of packets into the network every round-trip time (RTT), until finally after 5 RTTs (or 250ms), the window size plateaus at the maximum allowable size. At this point, TCP has achieved its goal and is fully utilizing the bottleneck link.

An important thing to note about Figure 2 is the correspondence between the queue length and changes in window size. Every time the TCP sender receives a single acknowledgment, it increases the window size by one, and injects two more packets into the network. Because acknowledgments return to the sender spaced apart by the bottleneck service time, so does each pair of packets. Because the bottleneck server only has time to process one of these packets before the next pair arrives, one packet accumulates in the queue for every pair that arrives. More generally, when the TCP window increases from W to $2 * W$ packets during one RTT, the maximum size of the queue for that time-interval will be W. If the maximum capacity of the network (and the maximum window size) of the network is W_{max} packets, then the bottleneck queue must be able to store at least $W_{max}/2$ packets. Many TCP receivers actually send "delayed acknowledgments" back to the sender, only acknowledging every other packet that they receive. Following the same logic, if the TCP receiver uses delayed acknowledgments, the bottleneck queue must be able to store at least $W_{max}/3$ packets in order to accommodate the sender's maximum window in a single burst.

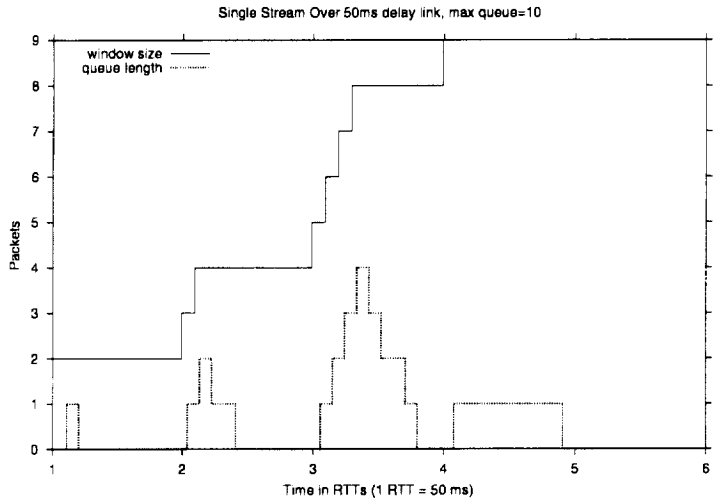

Figure 2: Window history of single TCP connection over a low-delay link. During its initial slow-start phase, the TCP window doubles every round trip time. Every time the window increases, TCP injects a window-sized burst of packets into the network. When the burst reaches the bottleneck router, the queue in the network builds up, and then drains before the next burst arrives. TCP stops increasing the window when it reaches the maximum window size of 9 packets.

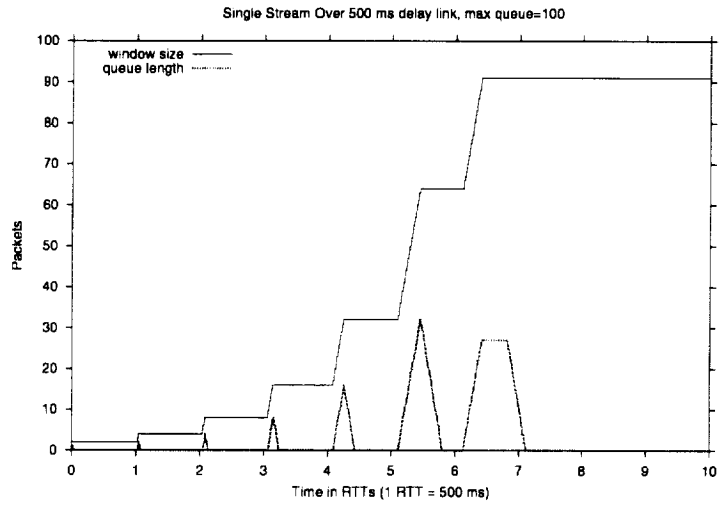

Figure 3: Window history of single TCP connection over a high-delay link. The history of this connection closely resembles the history of a low-delay connection, as depicted in Figure 2. However, because the delay of the network is higher, the total capacity of the network is also larger. TCP therefore takes more RTTs to ramp up to the maximum window than for a low-delay connection.

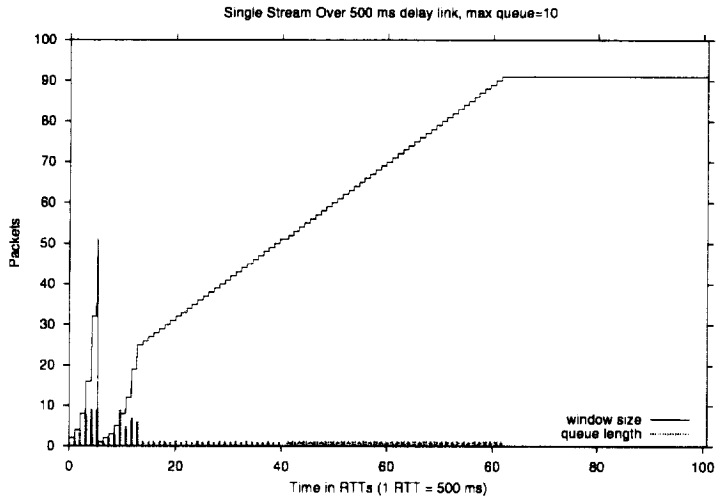

Figure 4: Window history of single TCP connection over a high-delay link with a short bottleneck queue. During slow-start, TCP overruns the bottleneck queue in the network before it is able to reach the maximum window size. TCP is only able to reach the maximum window size much later, during using congestion-avoidance.

In perfect conditions, TCP would achieve the performance depicted in 3. Many routers in typical networks are not configured to anticipate the large capacity of high-delay links, which is reflected in their relatively small queue lengths. Figure 4 shows another high-delay network with a 500ms RTT, but in this network the bottleneck queue length is 10 packets, much less than half the maximum capacity of the network. As before, when the TCP sender increases its window every RTT, it sends more and more packets into the network. However, in this case, the bottleneck server is not able to drain the queue as fast as it fills up, and the server drops packets that arrive when the queue is full. Once the TCP sender detects the lost packets, it reduces its window to 2 packets and initiates slow-start again. When it reaches half its previous maximum window, it exits slow-start and enters congestion-avoidance mode. In congestion-avoidance, the TCP sender increases its window by 1 every RTT, rather than doubling it. This strategy is reflected in a linear increase in the window size. After 100 RTTs (50 seconds), TCP finally reaches its maximum window of 91 packets. In fact, if the queues in the network are less than half the capacity of the network, the time it takes TCP to reach its full capacity will grow as a function of $O(RTT^2)$, significantly worse than the already poor growth of $O(RTT \log_2 RTT)$. If we had used a short TCP connection, like the Web connections that dominate Internet traffic today, in this example, the connection would have ended long before it had the chance to get off the ground.

Pacing is one possible solution to this problem [8]. The idea behind pacing is simple. Instead of sending an entire window of packets in a single burst at the beginning of each round-trip time, the TCP sender should send out the packets in a steady stream over the entire course of a round-trip time. For example, if the current TCP window is 4 packets and the RTT is 500ms, then the TCP sender should send out 1 packet every 125ms, instead of 4 packets every 500ms. An interval of 125ms is enough time for a 1.5Mb server to process a single packet, and therefore the queue in the network should never build up. In fact, in a paced TCP implementation, queueing bottlenecks should occur only when the TCP sender is genuinely sending at a rate that is too fast for the server itself. Pacing is an attractive solution to the queueing-bottleneck problem for at least two reasons. First, it relieves network designers of having to guess at buffer sizes based on typical round-trip delays. Second, it can be implemented by modifying TCP senders only. Pacing does not require participation of the network's intermediate routers or TCP receivers.

This rest of this paper examines TCP pacing in greater detail. Though we will draw examples from high-delay networks to illustrate queueing bottlenecks, it is important to keep in mind that queueing bottlenecks may occur in any network where the product of the delay and the bandwidth is high. The specific contribution of the delay in the product is not critical to causing a queueing bottleneck, it simply increases the cost when the bottleneck occurs. Section 2 of this paper discusses the issues involved in implementing TCP Pacing. Section 3 describes several TCP algorithms and their significance with regards to high-delay networks and TCP pacing. Section 4 presents TCP pacing performance results based on the ns network simulator. Finally, Section 5 gives an overview of the results and discusses possible directions for future work.

2 Pacing Implementation

Several factors are important to consider in implementing paced TCP. First is the need for modification of the TCP/IP protocol for high bandwidth-delay networks. For example, the TCP/IP sequence number space must be expanded to prevent the TCP/IP's sequence numbers from wrapping around within seconds in a high bandwidth-delay network. Second, paced TCP needs to use high-granularity timers in order to guarantee the release of a steady stream of packets. The granularity of this timer can have both positive and negative effects on the performance of paced TCP. In this section we discuss both modification of the TCP/IP protocol for high bandwidth-delay networks and the granularity of the timers used in pacing. In addition, we propose an implementation scheme for paced TCP based upon the leaky-bucket algorithm.

2.1 TCP/IP Protocol Modifications for High Bandwidth-Delay Networks

Several researchers have noted that the TCP/IP protocols now in use are not suited to high delay-bandwidth networks [9]. The TCP/IP packet headers contain several fields that TCP/IP agents use to keep track of their progress, and these fields simply cannot accommodate the large quantities involved in high delay-bandwidth connections. The inadequacy of these fields is the TCP/IP analogue of the Y2K problem. Fields that must be modified for high delay-bandwidth networks include IP Fragmentation, TCP sequence number, and TCP maximum window size. The IP Time to Live field must also be modified in networks where the delay component of the bandwidth-delay product is high. For the remainder of this paper, we will assume that paced TCP for high delay-bandwidth networks includes these modifications.

2.2 High-Granularity Timers and Pacing

A sender using paced TCP must send out a stream of packets at specific intervals of time, and a sender needs to use high-granularity timers to do this. It is possible to modify the granularity of operating system timers to do this, but the modifications carry consequences. Each TCP timer-handler incurs a significant overhead, so the more frequently TCP schedules timer interrupts, the more of an effect TCP pacing will have on the overall performance of the system. More specifically, given the following system parameters:

TCP window:	W (packets)
Round-trip time:	RTT (seconds)
TCP timer frequency:	$f = W/RTT$ (events/second)
Event handling cost:	c (cycle/event)
Machine Power:	P (cycle/second)

the system cost, SC, is given by the following equation:

$$\begin{aligned} SC &= f * c/P \rightarrow \\ SC &= \frac{W * c}{RTT * P} \end{aligned} \tag{1}$$

We can reduce the overall system cost by pacing the release of bursts of packets, rather than individual packets. Assuming that TCP sends out bursts of BST packets, we obtain the following modified equation for system cost:

$$\begin{aligned} f &= W/(BST * RTT) \rightarrow \\ SC &= \frac{W * c}{P * BST * RTT} \end{aligned} \tag{2}$$

It may be tempting to raise BST arbitrarily high to keep system costs to a minimum, but this approach can not be recommended. Every time TCP sends out a BST burst, $BST - 1$ packets pile up at the

bottleneck queue. We can characterize the load on the bottleneck queue, l, as follows:

$$l = f*(BST-1) = \frac{W}{RTT} - \frac{W}{RTT*BST} \quad (3)$$

Finally, if the maximum queue size in the network is q, then paced TCP should ensure that $BST < q$.

Timer granularity also has a direct effect on the rates at which TCP can send packets. Suppose that paced TCP uses a g = 1ms timer and BST = 1. TCP can send 1 packet every 1ms, every 2ms, every 3ms, etc. It can not, however, send out 1 packet every 1.5ms, every 4.5, or any other non-integer multiple of 1ms. This observation leads to the following equation, which relates the window sizes that TCP can support (W packets), the connection round-trip time (RTT seconds), and the granularity of the system timer (g seconds):

$$W = \frac{BST*RTT}{n*g}, n = 1,2,3,4.... \quad (4)$$

So, for instance, a paced TCP connection with RTT = 500ms, maximum timer granularity g = 1ms, and BST = 1 can support windows of sizes 500, 250, 125, 100, 83, and so forth.

2.3 A Leaky-Bucket Scheme for Paced TCP

In order to implement paced TCP, we propose using a modified *leaky-bucket scheme* for admitting packets into the network. In a typical leaky-bucket scheme, packets can only enter the network if they obtain a "token" from a bucket of tokens. If a packet wishes to enter the network and the bucket is empty, it must wait until a token becomes available in order to enter the network. There are two parameters that govern the behavior of a leaky-bucket flow: the maximum number of packets that the bucket can hold and the rate at which the bucket is replenished with tokens. By altering the size of the bucket we can prevent large bursts from entering the network. By limiting the rate at which we refill the bucket with tokens, we can limit the frequency with which these bursts can enter the network.

In paced TCP, just as in a leaky-bucket scheme, TCP tries to limit the size of bursts entering the network. However, TCP also needs to be able to modify the send rate of the leaky-bucket flow to mimic changes in its window. By making a couple of simple modifications to the standard leaky-bucket algorithm, we can implement paced TCP. First, a paced TCP refills its token bucket at dynamically changing intervals of time. So, for instance, imagine that the timer granularity is g = 1ms and BST = 4 packets. Then TCP's maximum bucket size will be 4 packets, and TCP will fill up the bucket with 4 tokens at time 0.000. Assume also that TCP's send rate (as determined by the current congestion window size and round-trip time) a time 0.000 is 1 packet/ms (or 4 packets every 4ms). TCP would then set its next bucket refill-time to occur at time .004 seconds. Then, at time .001 seconds, TCP receives an acknowledgment and doubles its send-rate to 2 packets/ms (or 4 packets every 2ms). TCP would then revise its next refill-time to occur at time .002 seconds from the original .004 seconds. This approach is aggressive in the sense that, when TCP's send-rate changes, the new rate is effective retroactively to the beginning of the current refill period. This aggressive approach allows TCP to adapt quickly to changes in its window even though it only refills the bucket at coarse-grained intervals.

To limit overhead, TCP keeps track of bucket refill-times using state variables and minimizes its use of timers to refill the bucket. If TCP has packets to send and it does not use up all the tokens in the bucket, it simply takes note of how many tokens are left, and when the bucket should next be refilled. The next time TCP has packets to send, it checks to see whether the last bucket refill-time has past, refills the bucket as necessary, and continues as before. The only time that TCP should use a timer to refill the bucket is when it uses up all the tokens in a bucket and has more packets to send. This approach takes advantage of the fact that TCP wakes up frequently on its own, due to returning acknowledgments, and that scheduling timers to wake up at these times is redundant and wasteful. During slow-start, TCP should be able rely on acknowledgments for approximately half of its refills, reducing the pacing overhead to 50% less than that given in Equation 2.

3 Experimental Algorithms

In order to study the effect of pacing on TCP, we examined two existing TCP algorithms and one proposed TCP algorithm that is not yet widely used. In addition, we discuss a technique that is especially well-suited for estimating window size in high bandwidth-delay networks.

3.1 Classic TCP

Classic TCP is TCP as it was originally proposed by Jacobson [3]. Classic TCP maintains four state variables to keep track of its progress: the current congestion window (W), the maximum congestion window (W_{max}), the current slow-start threshold (*ssthresh*), and an estimate of the current round-trip time (RTT). Using these variables, classic TCP alternates between three modes of operation:

- Slow-start. In this mode, TCP sends an exponentially increasing number of packets to the receiver every RTT. Starting with an initial congestion window of $W = 2$, the TCP sender increases the congestion by 1 every time it receives an acknowledgment, until it reaches either the maximum window, W_{max}, or the slow-start threshold, *ssthresh*. If it reaches *ssthresh*, TCP enters congestion-avoidance mode. If TCP detects that a packet has been lost in the network, it sets $ssthresh = W/2$, and enters exponential-backoff mode.

- Congestion-avoidance. In congestion-avoidance mode, TCP sends a linearly increasing number of packets every RTT. It does this by increasing its window by 1 every RTT. if The sender does not receive an acknowledgment for an outstanding packet within a certain amount of time (determined by the RTT), it sets $ssthresh = W/2$, and enters exponential-backoff mode. The sender never increases W past W_{max}.

- Exponential-backoff. In this mode the sender attempts to recover from loss. It repeatedly sends a copy of the last unacknowledged packet to the receiver, exponentially increasing the time between each attempt, until the receiver acknowledges the packet. Once the receiver acknowledges the packet, the sender goes back to slow-start mode.

In practice, existing TCP implementations almost always use more sophisticated congestion control algorithms than these. We use classic TCP in this study, however, because it provides us with a baseline for our comparisons.

3.2 TCP-Reno

TCP-Reno incorporates Jacobson's fast-retransmit and fast-recovery algorithms to improve upon the way that classic TCP recovers from individual packet losses [4]. In fast-retransmit, the TCP receiver repeatedly acknowledges the last in-order packet that it has received. When the TCP sender receives the duplicate acknowledgments, it infers that a packet was lost, and resends that packet. The advantage of this strategy is that the TCP sender can detect lost packets much sooner than it might otherwise using timeouts. If fast-retransmit successfully recovers a lost packet, TCP may infer from incoming acknowledgments a that only a few packets were lost. In this case, it may perform a fast-recovery, which means that it will immediately set its window to $W = ssthresh$ and enter congestion-avoidance mode. The advantage of this strategy is that it circumvents the expensive slow-start process in the cases where only a few packets in the network were lost. TCP-Reno is the most widely deployed variant of TCP used in the Internet today.

3.3 TCP-FACK

TCP-FACK takes an even more aggressive approach to loss-recovery by providing the receiver with an explicit means of describing losses [6]. When a TCP-FACK receiver receives non-contiguous data, it also sends duplicate acks back to the receiver, as is the case in TCP-Reno. These acknowledgments, called

SACKs, carry the same information as standard TCP acknowledgments, plus additional information explicitly listing the data that has been correctly received. When the TCP-FACK receiver receives these acknowledgments, it knows exactly which data has been lost, and which data should be resent. TCP-FACK senders can also use the data contained in SACKs to glean information about the congestion state of the network and then to make intelligent decisions about what data to resend and when. Though TCP-Fack has not yet been widely deployed in the Internet, preliminary simulation studies show that TCP-FACK can use available bandwidth at the bottleneck server more effectively than standard TCP implementations, leading to significant improvements in throughput performance.

3.4 Packet-Pair Probes

A TCP sender can estimate the service rate of the bottleneck link using a technique called *packet-pair probes* [5] [2]. In this technique, the sender sends out a pair of packets, back-to-back, and waits for their corresponding acknowledgments. When the bottleneck server receives the packets, it will process them sequentially, and they will exit the bottleneck server with a delay between them corresponding to the bottleneck service time. The end receiver will, in turn, acknowledge these packets as soon as they arrive. When the sender finally receives the acknowledgments, it can interpret the spacing between the packets to glean information about the bottleneck server. As noted in [5], the spacing of the packet-pair probes can have several meanings. If the bottleneck server serves packets in a first-in-first-out order, as is the case with most routers in the Internet today, then the interval will only reflect the maximum available bandwidth of the bottleneck server. If the bottleneck server serves packets in a round-robin order, as is the case in a fair-queueing router [1], then the interval reflects the fraction of the total bottleneck bandwidth currently available at the server.

It is easy to see why current TCP implementations do not use packet-pair probes: TCP contains its own mechanisms for estimating bandwidth and over the years these mechanisms have served their purpose perfectly well. However, there are a number of ways that packet-pair probes might be used to improve TCP's performance in a high bandwidth-delay network. First, under-estimating or over-estimating TCP's maximum window can adversely impact TCP's throughput performance, a problem that only worsens as the network delay increases. A TCP sender may be able to use packet-pair probes to set TCP's initial maximum window size, W_{max}. Second, in a system where the bottleneck server implements fair-queueing, TCP can actually use packet-probes to estimate the congestion window, W, itself.

4 Simulation Results

In this subsection we present the results of several simulations that we performed to evaluate the performance of TCP pacing.

4.1 ns Simulator

ns [7] is an event-driven network simulator with extensive support for simulation of TCP, routing, and multicast protocols. To implement TCP pacing, we extended ns's built-in TCP agents using the pacing algorithm described in the previous subsection, as well as added support for packet-pair probes. The behavior of the modified TCP agents can be customized by turning pacing on or off, setting the pacing burst size and the TCP timer granularity, and turning packet-probes on or off. This simulator produces several metrics that track the progress of TCPs:

- Congestion window. The size of TCP's congestion window, W, limits the number of packets that TCP can inject into the network at any time. The evolution of this window during the course of a TCP connection has a direct impact upon the total throughput of the connection.
- Packets in the pipe. In a standard TCP implementation, TCP attempts to keep the number of packets in the network pipe, i, always equal to the congestion window, W. If the window W jumps up suddenly, then TCP sends an instantaneous burst of packets into the network to match the change in the window W. It is exactly this behavior that a paced version of TCP attempts to avoid. By observing the evolution of i with respect to W, we can verify the correctness of our pacing algorithms.
- Bottleneck queue length. We expect to see a standard TCP implementation overrun the queues long before it reaches its maximum window. In a paced TCP implementation, however, we should not see the queues build up at all.
- Overall throughput. The overall throughput of a connection measures the number of packets that the TCP sender was able to successfully send to the receiver, during some fixed interval of time. Overall throughput can indicate the effectiveness of a TCP implementation at using the bandwidth available at the bottleneck.

A detailed description of the ns implementation of pacing is provided in Appendix A.

4.2 Single-Stream Pacing Performance

We began our study of TCP pacing by looking at the performance of classic TCP over a single bottleneck link. The network topology that we used for the simulations is given in Figure 1, and the the simulation parameters are given in Table 2. Figure 5 shows the results of this simulation using a paced, classic TCP sender. The paced TCP sender is able to smoothly able to increase its congestion window to the ideal, maximum window of 91 packets during slow-start, without encountering a false-bottleneck. The absence of the false-bottleneck can be explained by the fact that the bottleneck queue never grows beyond zero packets. As described in Equation 4, however, a TCP sender with a 1ms timer can only support window sizes of 83 or 100 packets. This explains why the number of packets in the pipe never reaches beyond 83 packets per RTT.

Parameter	Setting
Round-trip Delay	500ms
Bottleneck bandwidth	1.5 Mb
Queueing discipline	Drop-tail
Packet size	1024 bytes
Bottleneck queue length	10 packets
Timer granularity	1ms
Burst-size	1.2 packets
Network capacity/ TCP maximum window size	91 packets

Table 2: Simulation setup for a TCP connection running over a bottleneck link, as depicted in Figure 1.

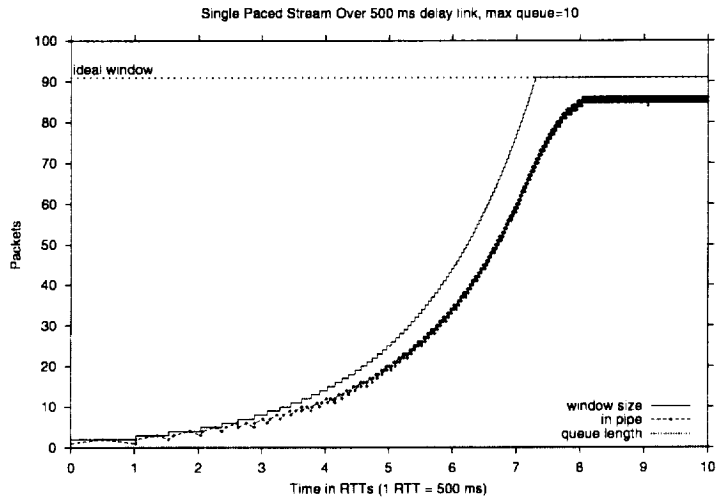

Figure 5: Queue size and window evolution for a single TCP connection running over a bottleneck link. The TCP congestion window is able to ramp up to the ideal size of 91 packets without encountering a false-bottleneck because the queues in the network never build up. The 1ms timer used in this simulation is only able to sustain a rate of 83 packets per RTT, less than the 91 packets per RTT that could be sustained by the full capacity of the network.

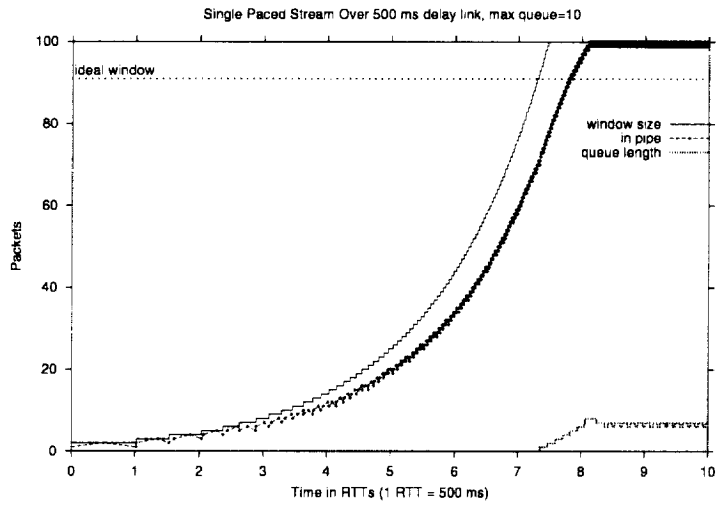

Figure 6: Paced TCP over a 1.5 Mb link with 500ms RTT and maximum window 100. The paced TCP sender is able to ramp all the way up to the maximum window size of 100 packets/RTT. This window size is larger than the ideal window size, causing packets to build up in the bottleneck queue.

Figure 6 shows the same simulation, except with the TCP sender maximum window set to 100. As this figure illustrates, this TCP sender is again able to smoothly able to ramp up to the maximum congestion window in a little over 7 round-trips. In contrast with Figure 5, the line representing the packets in the pipe does eventually rise to meet the line representing the congestion window. The penalty for increasing the maximum window to 100, beyond the ideal window of 91, manifests itself in the evolution of the bottleneck queue length. At 100 packets/RTT, the TCP sender sends packets to the receiver at a rate faster than the bottleneck server can sustain. Though it never actually exceeds the capacities of the queues, the queues stay almost constantly full at a level of 8 packets every RTT.

By increasing the size of the bursts paced out by TCP, we can exactly match the capacity of the network, rather than overutilizing or underutilizing it. Figure 7 shows the same simulation as Figure 5, this time with the TCP burst size set to 2 packets. Following Equation 4, a TCP sender with a .001 second timer and a burst size of 2 packets can sustain a windows of size 90 packets, close to the ideal window size of 91 packets. As the figure illustrates, the line representing the packets in the pipe again rises to meet the line representing the congestion window. This time, however, the queues in the network never build up to more than 1 packet. Figure 8 shows a magnified view of the first few RTTs of this connection. Instead of pacing out single packets, this connection paces out bursts of 2 packets. Every time one of these 2-packet bursts enters the network, the the first packet enters service, and the second packet waits in the queue for service. This behavior is reflected in the fact that the line representing queue length goes up to 1 every time the TCP sender sends out a burst of packets.

Table 3 summarizes these results.

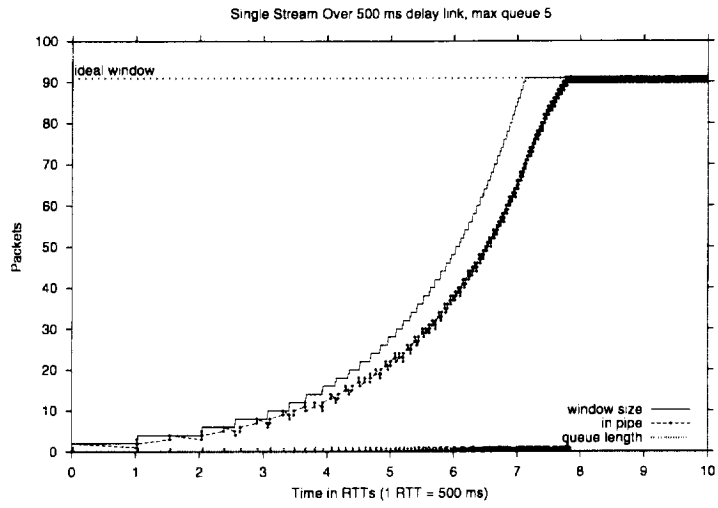

Figure 7: Paced TCP over a 1.5 Mb link with 500ms RTT and burst-size 2. The paced TCP sender is able to increase the congestion window to the ideal window size of 91 packets by pacing out bursts. Every time the TCP sender paces out a 2-packet burst, the bottleneck queue grows to 1 packet.

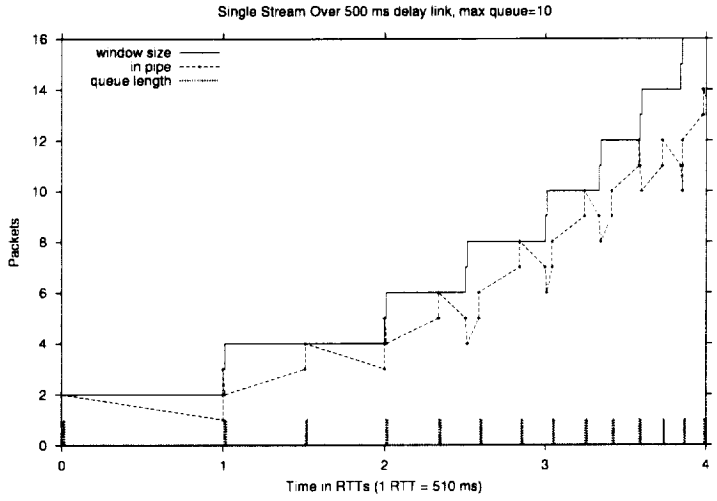

Figure 8: Magnified progress of paced TCP over a 1.5 Mb link with 500 ms RTT and burst-size 2. Every time the TCP sender paces out a 2-packet burst, the bottleneck queue grows to 1 packet. The number of packets in the pipe decreases every time the sender receives an acknowledgment for a packet, which accounts for the dips in the corresponding line in the graph.

Maximum window	Pacing	Queue length	Burst-size	Throughput	Average queue length (non-empty)
91	N	100	NA	1734	.2 (14.9)
91	N	10	NA	1450	.1 (1.3)
91	Y	10	1	1614	0 (0)
100	Y	10	1	1771	6.3 (6.6)
91	Y	10	2	1728	0 (1)
91 (with probes)	Y	10	2	1750	0 (1)

Table 3: Summary of results for single-link simulations. These results show that a paced TCP connection with burst-size 2 can achieve close to the same throughput as an unpaced TCP connection running over a bottleneck with large queues. The rightmost column describes the average length of the bottleneck queue over time. The average non-empty queue length appears in parentheses.

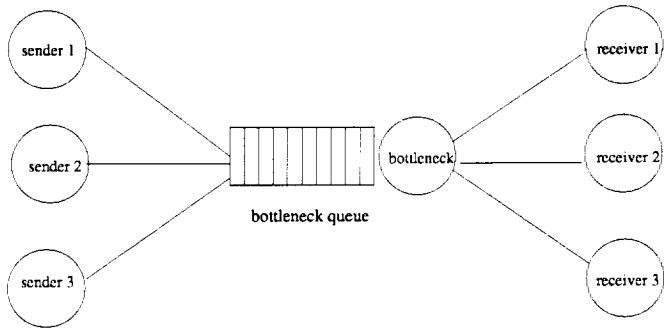

Figure 9: Network topology for multiple stream simulations.

Algorithm		Packets sent	Average queue length (non-empty)
Classic TCP	unpaced	36799	1.1 (2.6)
	paced	41065	9.8 (2.2)
TCP-Reno	unpaced	35711	1.0 (2.6)
	paced	39515	.86 (2.0)
TCP-FACK	unpaced	42750	1.4 (2.7)
	paced	45098	1.2 (2.3)

Table 4: Simulation results for different algorithms running over a shared bottleneck link. The rightmost column describes the average length of the bottleneck queue over time. The average non-empty queue length appears in parentheses.

4.3 Multiple Stream TCP Performance

For our second set of simulations, we examined the performance of paced TCP when several streams compete for access over the same bottleneck link. The topology we used for these simulations is depicted in Figure 9. We used the same set of parameters given in Table 2, setting the maximum window size for each TCP sender to 91 and the pacing burst-size to 2. We staggered the start times for each TCP connection, starting each connection 60 seconds after the last connection, and then letting each stream run for 180 seconds. We staggered the streams in order to force the TCP connections to adjust to both the arrival and relief of congestion[1]. We ran each simulation 20 times and averaged the results together to obtain our results.

Table 4 shows the simulation results for 3 different TCP algorithms, with or without pacing. The results show that pacing improves all three TCP algorithms, with classic TCP and TCP-Reno achieving the most dramatic improvements from pacing. TCP-Fack with pacing achieves the highest throughput of all the algorithms, a 1.23x improvement over unpaced, classic TCP.

4.4 Packet-Pair Probes

For our simulation, we examined the use of packet-pair probes in a paced TCP implementation. In this simulation, TCP uses packet probes to determine the bottleneck bandwidth during slow start, and instantaneously sets the congestion window to match the bottleneck rate. We ran this version of TCP

[1]During the course of our study, we discovered that TCP senders would synchronize themselves to the bottleneck server due to the lock-step nature of the simulator. As a result, the TCP connection that arrived first at the bottleneck would use the full capacity of the link and never yield to other connections. We introducing an infrequent, millisecond jitter into the bottleneck server to circumvent this problem.

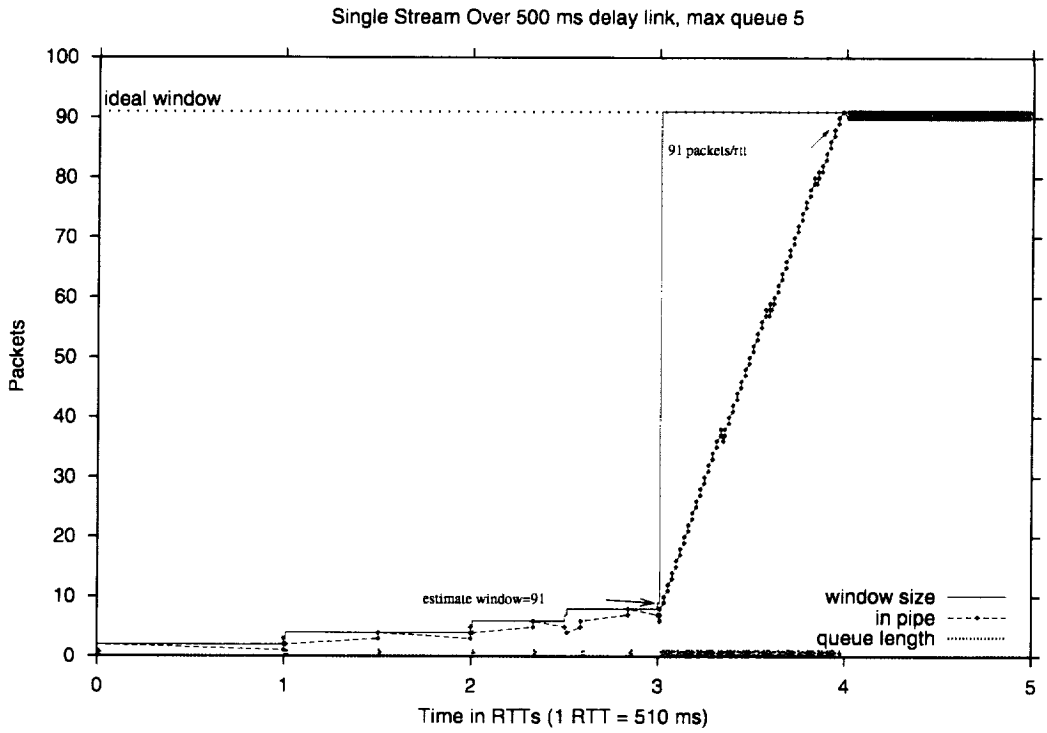

Figure 10: Evolution of a TCP connection that uses packet-probes to estimate the bottleneck bandwidth during slow-start. After 3 round-trip times, the TCP sender estimates the ideal congestion window to be 91 packets, and paces out 91 packets over the course of a single round-trip.

over the single-stream topology given in Figure 1, with the TCP burst-size set to 2 packets. Figure 10 depicts the results of this simulation. After 3 round-trip times, TCP is able to correctly estimate the bottleneck congestion window to be 91 packets, and sets the window to be 91 at that time. Because the system paces out its packets, the TCP sender then proceeds to pace out the 91 packets over the course of a single RTT, and remains at that window for the rest of the connection. Note that an unpaced version of TCP would respond to the increase in window size by instantaneously injecting 91 packets into the network. Since the bottleneck queue can only accommodate 10 packets, 81 of these packets would be dropped. A paced version of TCP, however, can accommodate such dramatic changes in the sender window size. A summary of the overall performance of this simulation appears in Table 3.

5 Discussion

In this paper, we described Paced TCP, a modified version of TCP that overcomes the problems of queueing bottlenecks in wireless, high bandwidth-delay networks. A standard TCP implementation encounters queueing bottlenecks when it sends more packets in a single burst than can be stored at the bottleneck queue. This problem is particular to high bandwidth-delay networks, because the queues in the network are disproportionate to the size of the bursts that TCP sends out. Paced TCP circumvents queueing bottlenecks by sending out frequent, small bursts where standard TCP would send out infrequent, large bursts. We have shown how it is possible to implement paced TCP using a modified leaky-bucket scheme and discussed some of the trade-offs involved in using timers to implement pacing. Finally, we have provided simulation data showing that pacing can significantly improve the performance of three different flavors of TCP, with TCP-FACK yielding a 1.22x performance improvement over unpaced, classic TCP.

Though this does paper does answer many questions about TCP pacing, it does raise others. One attractive feature of paced TCP is that it decouples TCP's algorithms from the length of the queues in the network. If all TCP implementations were paced, network designers would never have to guess at buffer sizes based on the vague, ever-changing characteristics of the entire network. Following this line of reasoning, however, TCP's current implementation of receiver flow-control makes little sense. At the beginning of every TCP connection, the TCP receiver tells the sender the number of packets it can handle in a single burst, the "maximum receive window". The TCP receiver may change its window dynamically, and the TCP sender must respect this window. As originally conceived, the receiver can use this window to prevent the TCP sender from sending too quickly and from overrunning its queues. Unfortunately, as is the case with routers within the network, TCP receivers often set the maximum receive window to a value that is disproportionately low for a high bandwidth-delay network, sometimes to as little as 5 to 12 packets. If a TCP connection sets its maximum congestion window to 5 packets and the ideal window for the network is 91 packets, the sender will be forced to grossly underutilize the capacity of the network. This window limitation seems particularly pointless since, as we have shown, a paced version of TCP can increase its window to 91 packets without loading up the queues in the network at all. Some compromise must exist between restricting the sender too much and not restricting it at all. Automatic TCP buffer tuning [10] and packet-pair probes are two techniques that may hold keys to overcoming this problem. We will continue pursuing this question in our future investigation of paced TCP.

References

[1] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, September 1990.

[2] Janey C. Hoe. Improving the start-up behavior of a congestion control scheme for tcp. In *ACM SIGCOMM*, 1991.

[3] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*. 1988.

[4] Van Jacobson. Modified congestion avoidance algorithm. Note to the `end2end-interest` mailing list. April 1990.

[5] Srinivasan Keshav. A control theoretic approach to flow control. In *ACM SIGCOMM*, 1997.

[6] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgment: Refining tcp congestion control. *ACM SIGCOMM*. 26(4), October 1996.

[7] ns-2 Network Simulator. http://www-mash.cs.berkeley.edu/ns/. 1998.

[8] C. Partridge. Ack spacing for high delay-bandwidth paths with insufficient buffering. End-To-End Research GroupRequest for Comments: DRAFT, September 1998.

[9] Craig Partridge and Timothy J. Shepard. Tcp/ip performance over satellite links. *IEEE Network*, September/October 1997.

[10] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic tcp buffer tuning. *ACM SIGCOMM*. 28(4), October 1998.

A Appendix: Detailed ns Implementation

The standard **ns** implementation includes a TCP "Agent" C++ class that simulates the actions of a TCP sender/receiver for a single flow at a single node. It also includes several sub-classes of TCP agents to simulate different variations of TCP, including TCP-Reno and TCP-FACK. In order to augment the standard **ns** TCP implementation with pacing, we created a new "Pacer" class that can be used by any TCP agent to implement pacing. The Pacer uses the following state-variables:

1. Maximum burst size. The maximum sized burst that we are allowed to send. This variable is also directly accessible from the Tcl interpreter.

2. Timer period. The granularity of the timer. The maximum rate at which we will be able to send packets will be be the maximum burst size divided by the timer period. This variable is also directly accessible from the Tcl interpreter.

3. Pacing timer. We set this timer to expire whenever we want to refill the bucket at some time in the future.

4. Last refill time. This is the time at which the bucket was refilled. The TCP flow is allowed to use up all the tokens in the bucket, but no more, until the next refill time.

5. Next refill time. This is the next time at which the flow's bucket will be refilled. This value is constantly recalculated based on TCP's dynamically changing send rate.

6. Tokens. The amount of tokens in the bucket. This value is invalid if the current time is greater than the next refill time.

7. Round-trip time. A cached copy of the last packet round-trip-time. This round-trip time is used to estimate the current send rate, which is then used to calculate the bucket refill times.

Typically, a paced TCP agent would create a new Pacer object at start-up, and then call the following functions, provided by the pacer:

1. `available_tokens`. This function calculates the amount of tokens currently available based on the beginning of the last bucket refill-time, the current time, the next refill-time, and the current send rate. If the Pacer has not yet past the next refill-time, this function returns the amount of tokens left within the bucket. If it has past the next refill-time, this function fills up the bucket, resets the last and next refill-times, and returns full tokens to the caller.

2. `adjust_tokens`. TCP calls this function whenever it sends out packets. First, this function will decrement the available tokens in the bucket by the number of packets just sent. If the caller indicates that it still has packets to send, this function will then schedule the timer to refill the bucket at the next refill time.

3. `expire`. This function is called whenever the pacing timer expires. This timer is set to expire at the next bucket refill-time, but only when there are packets awaiting tokens. This function calls the TCP function `send_much` to send out more packets.

4. `rtt_update`. The default TCP Agent calls this function whenever it gets a new packet with a new a new round-trip time (RTT). We modify this function to first cache the RTT before performing any additional processing.

In order to implement a paced version of ns's TCP base class, we simply modified one function, send_much. send_much is the function that the default TCP agent uses to send out a window of packets. At the beginning of the function, send_much calls the function available_tokens() to find out how many tokens are currently left in the bucket. Later, in the body of the loop that TCP usually uses to send out a window of packets, send_much adds an extra check to see whether it has reached its allotted amount of tokens. Finally, before exiting, send_much calls the function adjust_tokens to update the token count and schedule the pacing timer to expire, if necessary. We were able to implement paced versions of TCP-Reno and TCP-FACK by making analogous changes to these classes.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 2000	Final Contractor Report

4. TITLE AND SUBTITLE

A Simulation Study of Paced TCP

5. FUNDING NUMBERS

WU-632-50-51-00
N00600-92-C-3377

6. AUTHOR(S)

Joanna Kulik, Robert Coutler, Dennis Rockwell, and Craig Partridge

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

BBN Technologies
10 Moulton Street
Cambridge, Massachusetts 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

E-12041

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, Ohio 44135-3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR—2000-209416

11. SUPPLEMENTARY NOTES

Project Manager, William D. Ivancic, Communications Technology Division, NASA Glenn Research Center, organization code 5610, (216) 433-3494.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Categories: 03, 04, 17, and 32 Distribution: Nonstandard

This publication is available from the NASA Center for AeroSpace Information, (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

In this paper, we study the performance of paced TCP, a modified version of TCP designed especially for high delay-bandwidth networks. In typical networks, TCP optimizes its send-rate by transmitting increasingly large bursts, or windows, of packets, one burst per round-trip time, until it reaches a maximum window-size, which corresponds to the full capacity of the network. In a network with a high delay-bandwidth product, however, TCP's maximum window-size may be larger than the queue size of the intermediate routers, and routers will begin to drop packets as soon as the windows become too large for the router queues. The TCP sender then concludes that the bottleneck capacity of the network has been reached, and it limits its send-rate accordingly. Partridge proposed paced TCP as a means of solving the problem of queueing bottlenecks. A sender using paced TCP would release packets in multiple, small bursts during a round-trip time in which ordinary TCP would release a single, large burst of packets. This approach allows the sender to increase its send-rate to the maximum window size without encountering queueing bottlenecks. This paper describes the performance of paced TCP in a simulated network and discusses implementation details that can affect the performance of paced TCP.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Protocol; TCP; Satellite	26
	16. PRICE CODE: A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	